Machine Learning I
Week 4: Neural Networks

Martin Felder, Christian Osendorfer

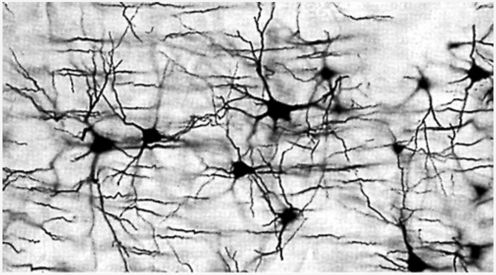
Technische Universität München

12 November 2009

Biological Neural Networks

A biological neural network consist of a collection of neurons linked together in a certain way, often (but not always) in layers.

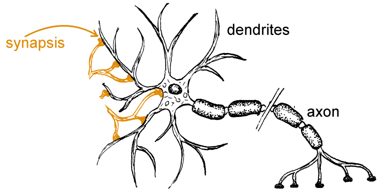
Biological neural network: Stained brain cells under the microscope



Biological Neurons

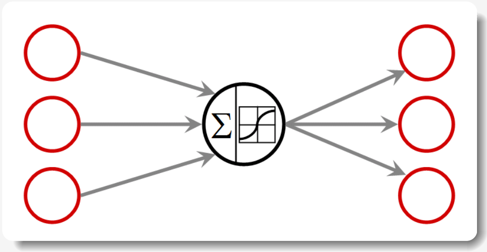
Each cell collects incoming electrical signals through its dendrites, processes their accumulated effect in a fairly simple manner, and forwards another electrical signal through its axon to a number of follow-on neurons, to which it is connected via synapses of varying conduction efficiency.

Biological Neuron



Simulated Neural Networks

A simulated neural network contains the same elements:

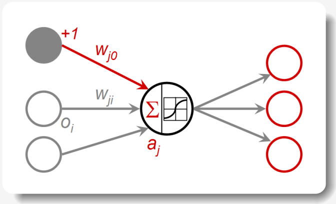


Neurons	sum up inputs and put them through a transfer function.
Synapses	are represented by real number weights, which modify the signals.
Axons/Dendrites	are wired together such that a structured system arises (often layered).

Simulated Neurons

Let w_{ji} be the weight associated with the synapse connecting neurons i and j of an arbitrary simulated network. Then, the activation a of neuron j is given by

$$a_j = \sum_{i=1}^{L} w_{ji} o_i + w_{j0}. \qquad (1)$$

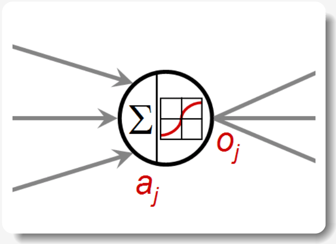


Here, o_i is the output of neuron i, there are L neurons connected to neuron j, and w_{j0} is the bias term of neuron j.
Bias is often included in the sum by defining an "on-neuron" with $o_0 \equiv 1$, which is assumed connected to all neurons in the network.

The Transfer Function

The response of a simulated neuron to its activation a_j is given by a transfer function

$$o_j = f_j(a_j). \qquad (2)$$



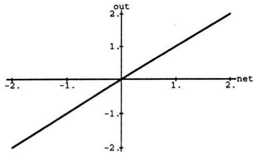
In the simplest case, the transfer function can be selected to be the identity function, or another linear function. However, in order for the network to exhibit nonlinear properties, $f(\cdot)$ also has to be nonlinear.

The Transfer Function

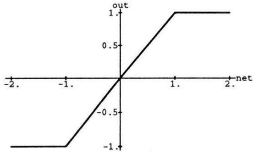
- Historically, binary and piecewise linear transfer functions were chosen first, in reference to the threshold-like behaviour of biological neurons.
- Smooth, S-shaped (=sigmoid) functions behave like threshold or linear functions, depending on scaling, and can be differentiated.
- Another network class, Radial Basis Function networks, uses Gaussian-shaped transfer functions.

$\longrightarrow$ compare with options discussed in the context of Linear regression!

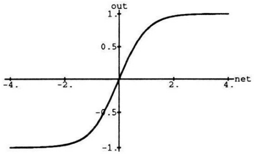
Commonly used Transfer Functions



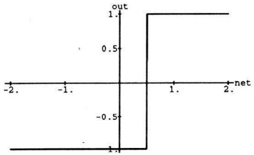
linear



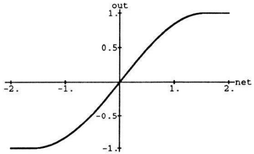
piecewise linear



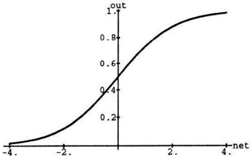
tanh(x)



threshold



sin(x) till saturation



logistic 1/(1+exp(-x))

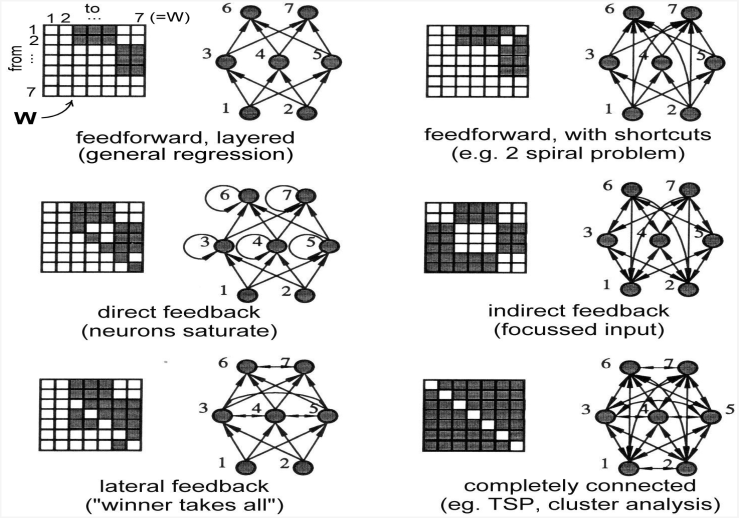
Network Architectures

Let $\mathbb{M}$ be a set of W neurons. The neurons are connected with weights $\mathbf{W} \in \mathbb{R}^W \times \mathbb{R}^W$. $K \leq W$ neurons shall be designated input neurons, ie. their outputs o_i are initialized with K input data values. Subsequently, the output of all W neurons can be calculated iteratively as

$$o_j = f_j(a_j) = f_j\left(\sum_{i=1}^{W} w_{ji} o_i + w_{j0}\right) \tag{3}$$

In practice, matrix $\mathbf{W}$ is often sparse; Its structure defines the network architecture and with it the kind of problems it can be best applied to:

Network architectures



Types of Neural Networks Covered

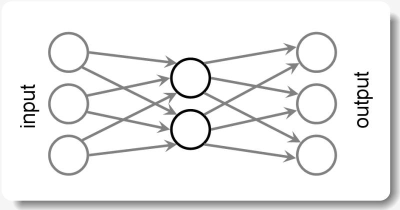
1. Multi-layer perceptrons (feed-forward; a workhorse of ML)
2. Self-organizing maps (special case; unsupervised)
3. Recurrent networks (later – need some other concepts first)

Multi-Layer Perceptrons (MLPs)

In our context, the following definition will be used:

MLP

A multi-layer perceptron is a general feed-forward network with differentiable, nonlinear transfer functions and at least two layers of weights between input and output.



Properties

Universal Approximation

An MLP with at least two layers of weights can approximate arbitrarily well any given mapping from one finite input space to another, given the number of hidden neurons H is large enough [eg. Hornik et al. 1989].

Scaling Properties

While the approximation error of classical approximation methods with fixed basis functions (linear, trigonometric, etc.) grows exponentially with the dimension of the problem, it grows only linearly for MLP-type regressions [Baron, 1993; Tresp 1995].

However...

Parameter Finding

Finding the optimum set of W weights for an MLP is an NP-complete problem, ie. it cannot be solved exactly within a time $t \propto W^x$ [Blum et al., 1992].

$\Longrightarrow$ need an iterative algorithm to find a set of weight as close to the optimum as possible.

Error Function

Recall from the Bayes regression lectures that we assume a functional correspondence $y(\cdot)$ between targets t and inputs x. The (measured) set $\{x^n, t^n\}$ will contain samples which satisfy

$$t = y(x, \mathbf{w}) + \epsilon. \tag{4}$$

Assuming Gaussian errors ϵ with standard deviation σ, i.e.

$$p(\epsilon) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\epsilon^2}{\sigma^2}\right), \tag{5}$$

the likelihood for each measurement n becomes

$$p(t^n|x^n, \mathbf{w}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y(x^n, \mathbf{w}) - t^n)^2}{\sigma^2}\right). \tag{6}$$

Quadratic Error Function

Again, we define the likelihood function

$$\ell = \prod_n p(t^n | x^n, \mathbf{w}) \tag{7}$$

Using the log likelihood trick again to obtain an error function $E \propto -\ln \ell$ yields

$$E \propto \sum_n (y(x^n, \mathbf{w}) - t^n)^2. \tag{8}$$

The proportionality constant again does not contain any $\mathbf{w}$'s, and thus is irrelevant for the minimization process. For convenience, one normally chooses it to be $\frac{1}{2}$, hence

$$E = \frac{1}{2} \sum_n (y(x^n, \mathbf{w}) - t^n)^2. \tag{9}$$

Quadratic Error Function

Generalization to multidimensional variables is straightforward and leads to the generic form of the familiar quadratic error function for independent variables

$$E = \frac{1}{2} \sum_n \|\mathbf{y}(\mathbf{x}^n; \mathbf{w}) - \mathbf{t}^n\|^2. \tag{10}$$

Further generalization is possible by allowing the variances to be different for each random variable, and correlated with covariance matrix $\mathbf{S}$:

$$E = \frac{1}{2} \sum_n (\mathbf{y}(\mathbf{x}^n, \mathbf{w}) - \mathbf{t}^n)^\mathsf{T} \mathbf{S} (\mathbf{y}(\mathbf{x}^n, \mathbf{w}) - \mathbf{t}^n). \tag{11}$$

Training Algorithms

Assume there is a set of paired input and target vectors, $\{\mathbf{x}^n, \mathbf{t}^n\}$, and we want to construct an MLP with output $\mathbf{y}(\mathbf{x}^n, \mathbf{W})$, to model the conditional probability function $p(\mathbf{t}^n|\mathbf{x}^n)$ according to the maximum likelihood principle, ie. such that the error function

$$E = \frac{1}{2}\sum_n \|\mathbf{y}(\mathbf{x}^n, \mathbf{W}) - \mathbf{t}^n\|^2. \tag{12}$$

is minimized. This can be achieved by demanding

$$\frac{\partial E}{\partial \mathbf{W}} = 0 \qquad \text{i.e.} \qquad \frac{\partial E}{\partial w_{ij}} = 0, \quad \forall w_{ij} = (\mathbf{W})_{ij} \tag{13}$$

Training Algorithms

Since our model is a feed-forward network, $\mathbf{y}(\mathbf{x}^n; \mathbf{W})$ is an explicit function of the input and the weights. Furthermore, because we demanded the transfer functions to be differentiable, the first derivative $\frac{\partial E}{\partial \mathbf{W}}$ can be calculated analytically. We could therefore apply a standard gradient descent algorithm to find the minimum of E, and thus the optimum coefficients (weights) $\mathbf{W}$. However, each calculation of a $\frac{\partial E}{\partial w_{ij}}$

requires at least one evaluation of the network model $\mathbf{y}$, and thus $\mathcal{O}(W)$ operations, where W is the number of adjustable weights in the network. Hence calculating $\frac{\partial E}{\partial \mathbf{W}}$ scales as $\mathcal{O}(W^2)$.

But: We can do better than that! How? $\longrightarrow$ Come back tomorrow!

Backpropagation (BP)

An elegant way of finding the error function's minimum is to take into account the special network topology by using the so-called error backpropagation (BP) algorithm.
It is obvious that E consists of a sum of independent terms E_n for each of the N training patterns, and so does its derivative, ie.

$$\frac{\partial E}{\partial \mathbf{W}} = \sum_n \frac{\partial E^n}{\partial \mathbf{W}} = \sum_n \frac{\partial E^n}{\partial \mathbf{a}^n} \frac{\partial \mathbf{a}^n}{\partial \mathbf{W}}. \tag{14}$$

Here, the chain rule was applied in the last step, with $\mathbf{a}^n$ being the activation of all neurons in the network. Now it is worth recalling that the weight matrix $\mathbf{W}$ for an MLP is actually very sparse, and it pays off looking at Eqn. 14 component-wise.

Backpropagation (BP)

In particular, we have

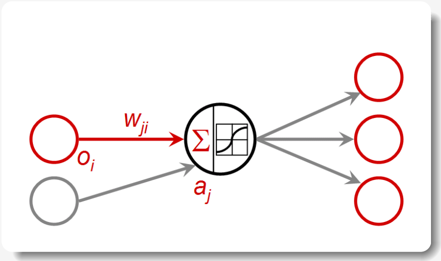
$$a_j^n = \sum_i w_{ji} o_i^n \qquad \text{and thus} \qquad \frac{\partial a_j^n}{\partial w_{ji}} = o_i^n \tag{15}$$

Furthermore, to simplify the notation we shall define

$$\delta_j^n \equiv \frac{\partial E^n}{\partial a_j^n}. \tag{16}$$

With this, it follows that

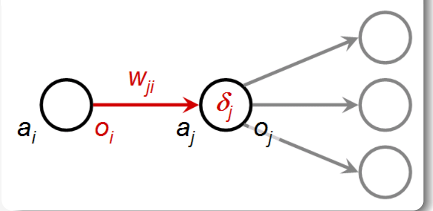
$$\frac{\partial E^n}{\partial w_{ji}} = \delta_j^n o_i^n. \tag{17}$$



Backpropagation (BP)

In other words, the derivative for each weight is calculated by multiplying the output o_i^n of the neuron at the front end of the connection with some "error factor" δ_j^n at the back end of the connection.

error backpropagation



Biases can be taken care of in the same manner by setting their output to $o_0^n = 1$ for all patterns n.

Backpropagation (BP)

The outputs of all neurons are calculated simply by forward-propagating the input vector n through the network. Then,

- for each output neuron k, $y_k^n = o_k^n = f(a_k)$, and thus

$$\delta_k^n = f'(a_k^n)\,(y_k^n - t_k^n). \tag{18}$$

- for each hidden neuron j we have $a_k^n = \sum_k w_{kj}\, f(a_j^n)$, thus

$$\delta_j^n = \frac{\partial E^n}{\partial a_j^n} = \sum_k \frac{\partial E^n}{\partial a_k^n}\,\frac{\partial a_k^n}{\partial a_j^n} = \sum_k \delta_k^n\, w_{kj}\, f'(a_j^n). \tag{19}$$

In particular, if neurons k in the output layer feature an identity transfer function, Eqn. 18 simplifies to

$$\delta_k^n = y_k^n - t_k^n. \tag{20}$$

CogBotLab Machine Learning & Cognitive Robotics

Backpropagation (BP)

This algorithm needs $\mathcal{O}(W)$ operations for calculating all outputs o_i, the same for backpropagating the errors δ_j and another $\mathcal{O}(W)$ operations to calculate the $\frac{\partial E^n}{\partial w_{ji}}$ from Eqn. 17. Thus the overall computational cost is still $\mathcal{O}(W)$, and not $\mathcal{O}(W^2)$ as with the generic approach.

Once the δ^n are have been calculated, the weights can be updated to decrease the value of the error function. In it simplest form, the BP algorithm uses online fixed-step gradient descent on the error surface E:

$$\Delta w_{ji}^{(n)} = -\eta \frac{\partial E^n}{\partial w_{ji}} = -\eta \delta_j^n o_i^n. \tag{21}$$

A (not too) simple example

As a practical example, BP now is to be applied to an atmospheric science data set. Suppose ozone number density ω at 25 km height is to be estimated from a dataset containing $N =$6371 sets of

- day-of-year
- latitude, longitude
- total ozone and
- 8 spectral channels from a satellite instrument

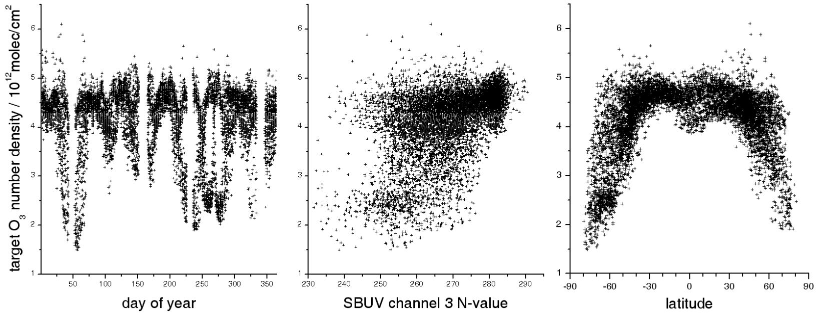
as input vectors $\mathbf{x}^n$, and

- ozone densities ω at 25 km from other measurements

as target "vectors" t^n. This data is for the year 1989.

Sample scatterplots x_i vs. t

As can be seen, the relationship between input and target data is highly nonlinear. It contains several different sources of noise (collocation error, measurement errors, retrieval errors, ...)



Network Setup

The basic MLP used for this experiment has

- 12 input neurons (linear transfer function),
- one hidden layer with 15 neurons (logistic transfer function), and
- one output neuron (linear transfer function).

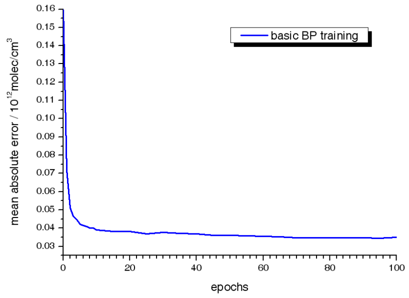
Input and output are being normalized.

Weights are initialized randomly.

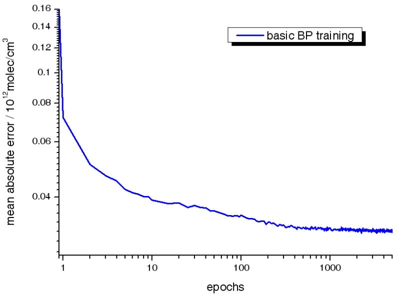
Example Results (I)

Training is performed for 5000 epochs with the simple BP algorithm introduced above.

Reduction of output error for the first 100 epochs.



Same but for all epochs (note double-log scale).



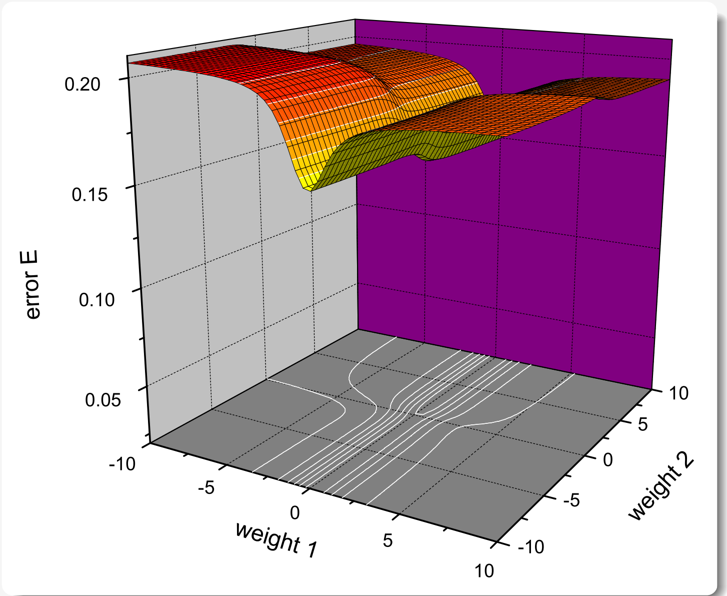
How does the error surface look like?

The following plot shows the development of the error surface E for two weights from the given example network during 1000 epochs τ of BP training:

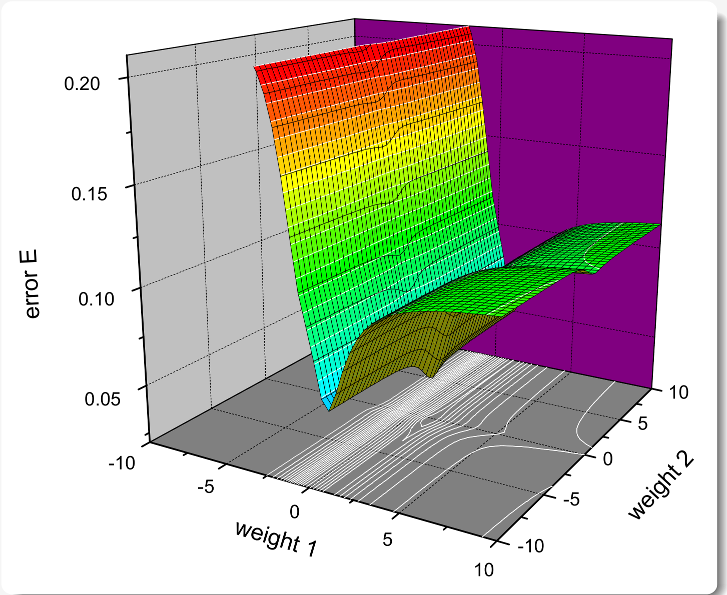
$$z(w_1, w_2, \tau) = E^{(\tau)}_{\mathbf{W}\setminus\{w_1,w_2\}}(w_1, w_2) = \frac{1}{2}\sum_n \|\mathbf{y}(\mathbf{x}^n; \mathbf{W}) - \mathbf{t}^n\|^2$$

It is clear to see that this surface is made from several overlapping sigmoid functions, with plateaus being generated where those sigmoids are in saturation. A distinct minimum is present, which is where the given weights reside.

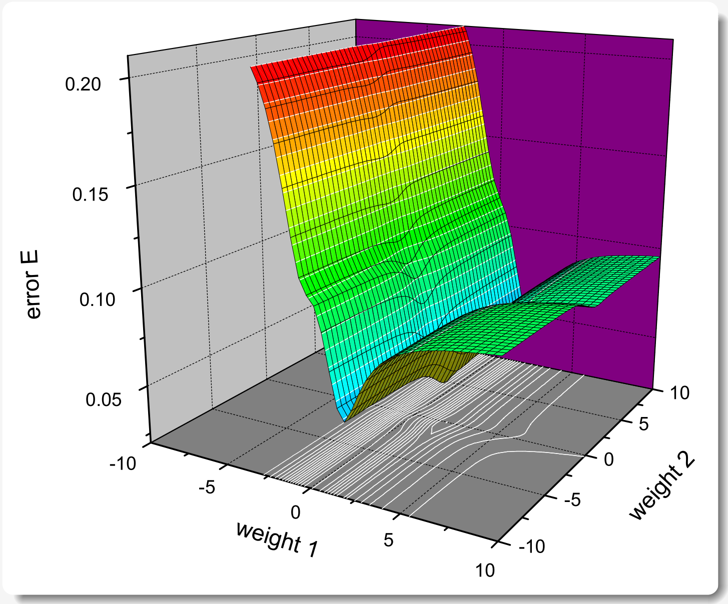
Due to the dependence of this error surface on all other weights in the network, its shape constantly changes; In particular, it tends to become lower and less steep.



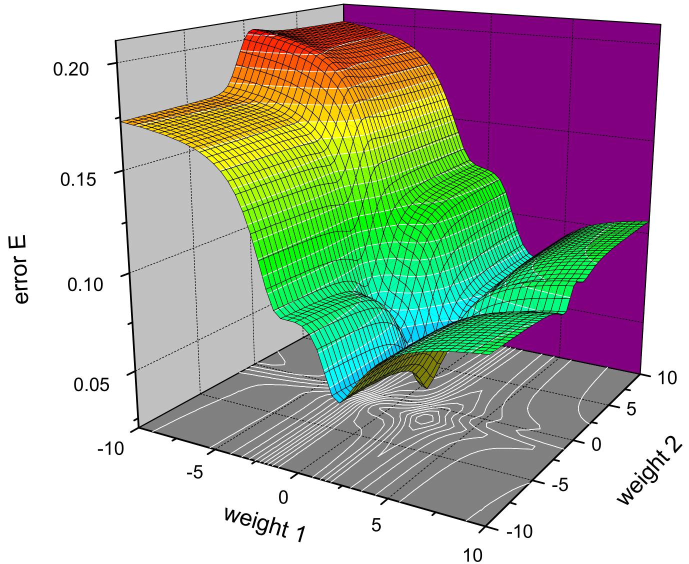
Epoch
0



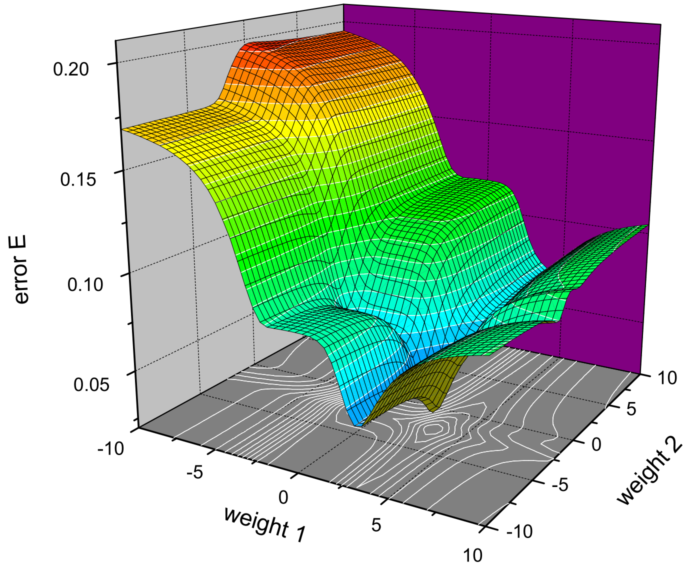
Epoch
1



Epoch
10

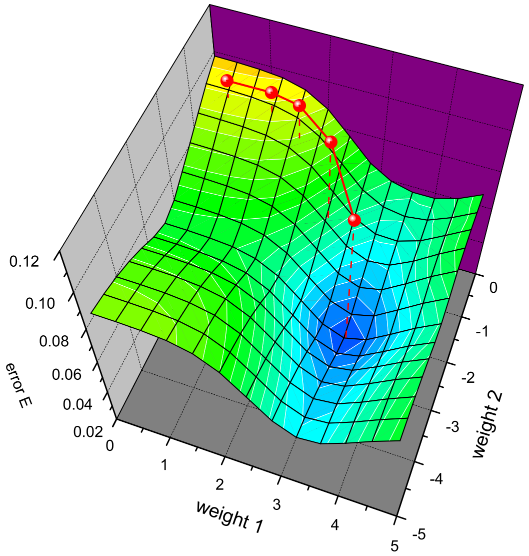


Epoch
100



Epoch
1000

Gradient descent on the error surface



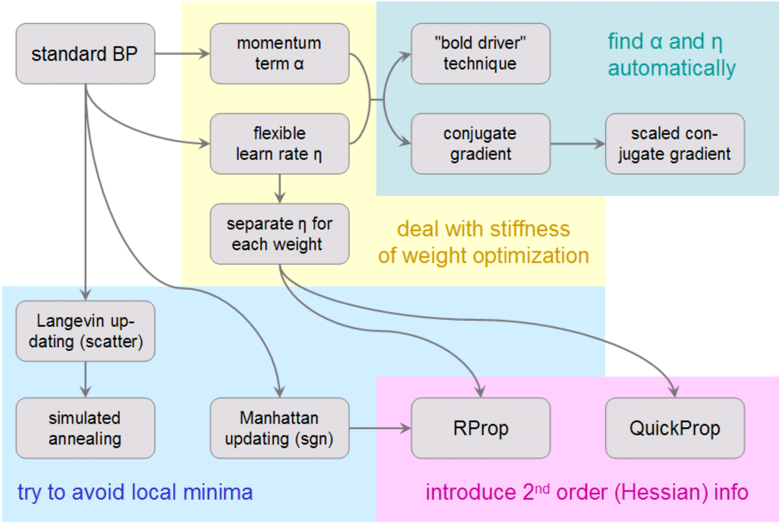
Close-up view of the minimum. The dots give the weights' values after 0, 1, 10, 100 and 1000 epochs of training.

Numerous other NN applications

- financial prediction
- mechanical fault detection
- medical classification (e.g. fMRI)
- betting (e.g. horse racing)
- house price estimation
- optical character recognition (OCR)
- speech transcription

(some of these are best done using recurrent networks... $\longrightarrow$ later!)

Heuristic methods to solve typical BP problems


standard BP
momentum term α
flexible learn rate η
separate η for each weight
"bold driver" technique
conjugate gradient
scaled conjugate gradient
find α and η automatically
deal with stiffness of weight optimization
Langevin updating (scatter)
simulated annealing
Manhattan updating (sgn)
RProp
QuickProp
try to avoid local minima
introduce 2nd order (Hessian) info

Momentum Term

For simplicity of notation, consider all W relevant elements of the weight matrix $\mathbf{W}$ compiled into a vector $\mathbf{w} \in \mathbb{R}^W$. Then the standard BP weight update Eqn. 21 becomes

$$\Delta \mathbf{w}^{(\tau)} = -\eta \frac{\partial E^{(\tau)}}{\partial \mathbf{w}} = -\eta \nabla E^{(\tau)}. \tag{22}$$

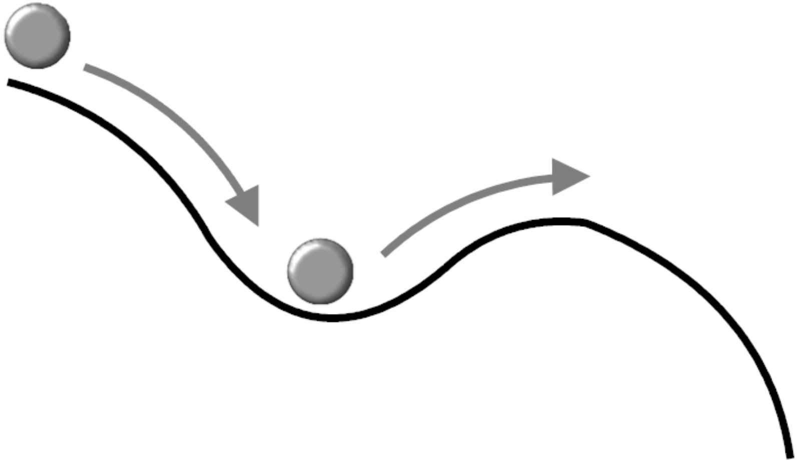
The introduction of a momentum term yields faster convergence (larger steps) in relatively flat areas of the error surface, while not hampering descent into steep valleys:

$$\Delta \mathbf{w}^{(\tau)} = -\eta \nabla E^{(\tau)} + \alpha \Delta \mathbf{w}^{(\tau-1)} \tag{23}$$

(usually choose either $\alpha = 0.1$ or $\alpha = 0.9$)

Momentum Term

Analogy to a ball rolling down a hill



Flexible Step Width

It is to be expected that once the optimization closes in on the minimum, the remaining adjustments have to become smaller to avoid overshooting and oscillations. The simplest way to do this is to set

$$\Delta \mathbf{w}^{(\tau)} = -\eta^{(\tau)} \nabla E^{(\tau)} \qquad \text{with} \qquad \eta^{(\tau)} = \eta^{(0)}\, r^{\tau} \tag{24}$$

The shrinkage factor r is typically very close to unity (0.999 in the sample problem). More sophisticated schemes like eg. the bold driver technique try to adjust η automatically based on convergence speed. Most of these schemes can be combined with the momentum term, though this does not automatically guarantee an improvement.

Separate step width for each weight

Stiffness results if the error responds on different scales for different weights, eg. due to widely varying sensitivity of the network output to its different input parameters – a highly likely situation for any reasonably complex training data. By setting

$$\Delta w_i = -\eta_i \nabla_i E, \tag{25}$$

and allowing the η_i to vary independently, the stiffness problem can be ameliorated. However, it is now unavoidable to combine this with an automated way of calculating the η_i, eg. the abovementioned bold driver scheme.

Avoidance of Local Minima

Langevin updating basically adds a Gaussian noise term ϵ to the weight updates, the amplitude of which exponentially decreases as training progresses. It is hoped that this will help the optimization step out of local minima.

$$\Delta w_i = -\eta \nabla_i E + \epsilon \tag{26}$$

A slightly more advanced form of this algorithm is simulated annealing, where the Gaussian noise is associated with a "temperature" and decreased in a physically consistent manner, just as real-world annealing lets eg. a piece of iron find its lowest energy configuration.

Resilient Propagation (RProp)

RProp [Riedmiller et al., 1993] uses only the sign of the gradient, but assigns each weight its own step width based on convergence behaviour. It turns out to be a very stable (hence the name) second order scheme. Let

$$\Gamma_i^{(\tau)} = \frac{\partial E^{(\tau)}}{\partial w_i} \cdot \frac{\partial E^{(\tau-1)}}{\partial w_i} \tag{27}$$

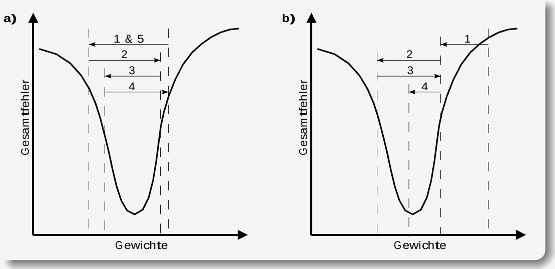
and then set

$$\Delta w_i^{(\tau)} = \begin{cases} \operatorname{sgn}\left(\frac{\partial E^{(\tau)}}{\partial w_i}\right) \cdot \Delta_i^{(\tau)} & \text{if} \quad \Gamma_i^{(\tau)} \geq 0 \\ -\Delta w_i^{(\tau-1)} & \text{if} \quad \Gamma_i^{(\tau)} < 0 \end{cases} \tag{28}$$

Resilient Propagation (RProp)

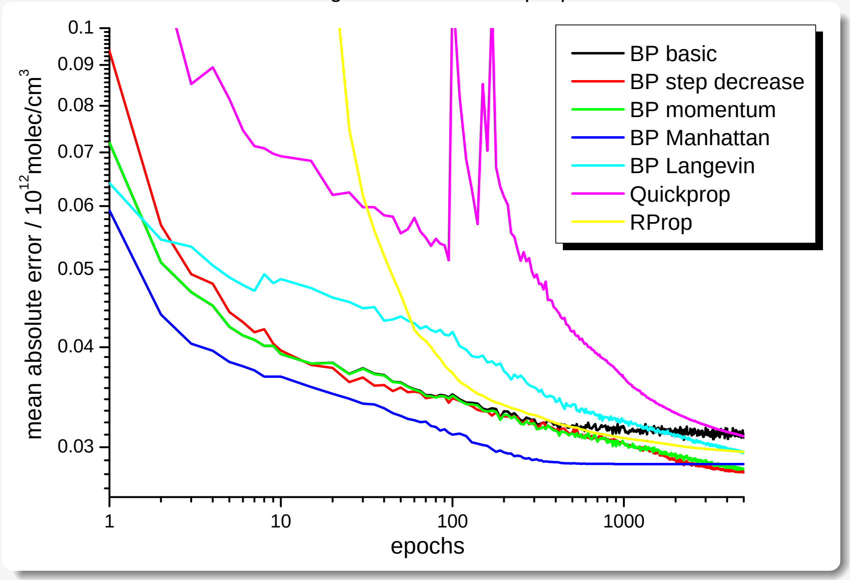
Here, Δ_i is the individual step width of weight w_i. This step width does not directly depend on the gradient, but is instead adjusted according to

$$\Delta_i^{(\tau)} = \begin{cases} \Delta_i^{(\tau-1)} \cdot \eta^+ & \text{if} \quad \Gamma_i^{(\tau)} > 0 \\ \Delta_i^{(\tau-1)} \cdot \eta^- & \text{if} \quad \Gamma_i^{(\tau)} < 0 \\ \Delta_i^{(\tau-1)} & \text{if} \quad \Gamma_i^{(\tau)} = 0 \end{cases} \qquad (29)$$



Mostly, one uses η^+=1.2 and η^-=0.5; RProp is not sensitive to those parameters.

Training error for the example problem



Classification Problems

Recap: In the context of Probabilistic Discriminative Models for two classes $\mathcal{C}_1, \mathcal{C}_2$ we introduced logistic regression as

$$p(\mathcal{C}_1|\phi) = y = \sigma(\mathbf{w}^T \phi(\mathbf{x})) \tag{30}$$

$$p(\mathcal{C}_2|\phi) = 1 - y, \tag{31}$$

with the logistic function σ defined as

$$\sigma(x) = \frac{1}{1 + \exp(-x)}.$$

Similarly, for an MLP we can do the same simply by using σ as transfer function in the output layer:

$$p(\mathcal{C}_1|\mathbf{x}) = y = \sigma(a) \tag{32}$$

$$p(\mathcal{C}_2|\mathbf{x}) = 1 - y \tag{33}$$

CogBotLab Machine Learning & Cognitive Robotics

Classification with MLPs

If we use a coding scheme for the target classes t_n where $t_n = 1$ if the correct class is $\mathcal{C}_1$ and $t_n = 0$ if the correct class is $\mathcal{C}_2$, we can combine the above expressions to write

$$p(t_n|\mathbf{x}) = y_n^t(1-y)^{1-t_n} \tag{34}$$

Assuming independent observations $n = 1, \ldots, N$ results in the binary cross entropy error function

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\{t_n \ln y_n + (1-t_n)\ln(1-y_n)\}$$

where $\mathbf{w}$ are our parameters, i.e. this time the weights of the network.

Multiclass Problems

Having $K > 2$ classes, use K output units in a 1-of-K coding scheme, and normalise output activations with the softmax function:

$$p(\mathcal{C}_k|\mathbf{x}) = y_k = \frac{e^{a_k}}{\sum_{k'=1}^{K} e^{a_{k'}}}, \tag{35}$$

The corresponding cross-entropy error function is then

$$E = \sum_n E_n = -\sum_n \sum_{k=1}^{K} t_{kn} \ln y_{kn}, \tag{36}$$

However, it turns out this more complicated error function (compared to regression) cancels out with the derivative of the softmax function, such that still

$$\delta_k^n \equiv \frac{\partial E_n}{\partial a_k} = t_{kn} - y_{kn}. \tag{37}$$

Error and Output Activation Functions

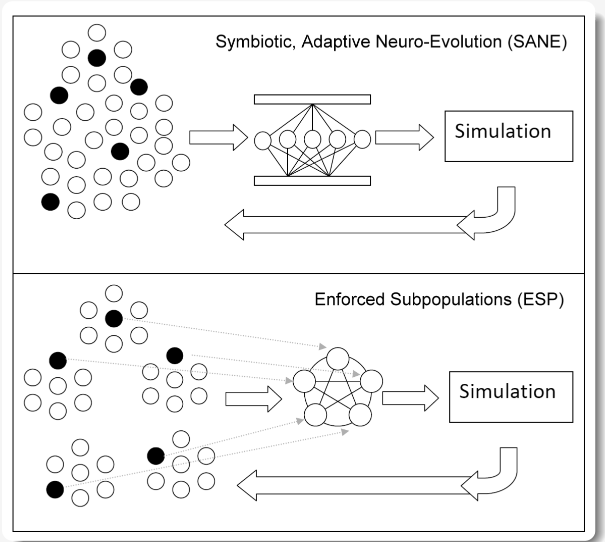
Note that for the purpose of back-propagating the errors, there is a "natural" correspondence of output activation and error function:

- For regression use *linear* outputs and *quadratic* error function
- For 2-class problems use *logistic* outputs and *cross-entropy* error function
- For multiclass problems use *softmax* outputs and *cross-entropy* error function

Each of these leads to the above simple form of the deltas, namely

$$\delta_k^n \equiv \frac{\partial E_n}{\partial a_k} = t_{kn} - y_{kn}.$$

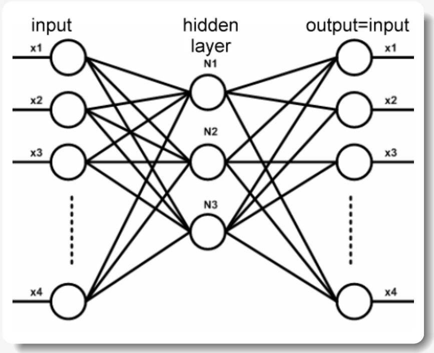
Outlook: Neuroevolution (later in the lecture)


Symbiotic, Adaptive Neuro-Evolution (SANE)
Simulation
Enforced Subpopulations (ESP)
Simulation

Autoencoder networks

Idea: Find compact representation of inputs (unsupervised!) by

1. letting the network re-create its own inputs, i.e. $\mathbf{t}_n \equiv \mathbf{x}_n$, and
2. creating a bottleneck by using fewer hidden units than inputs.



- Activations of hidden units $=$ compact code for $\mathbf{x}$.
- Usually many ways of encoding the same input and output.
- All inputs should be normalized.

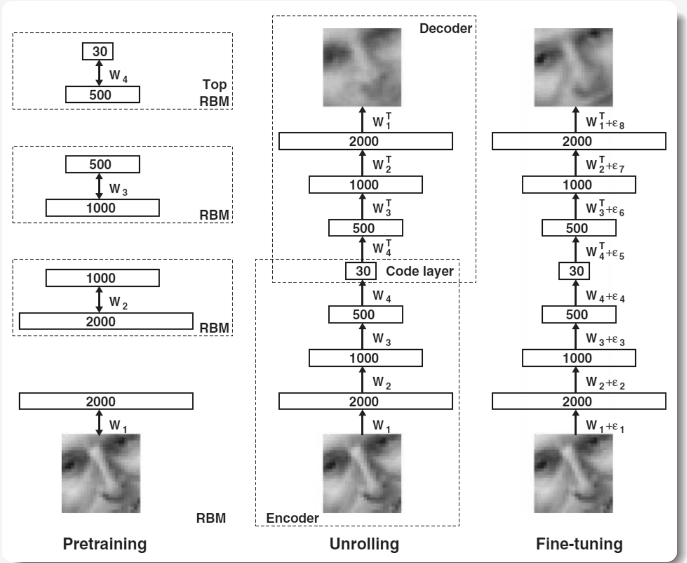
Deep Autoencoders

Idea: Brain uses many neuron layers to compress data into compact codes!
$\longrightarrow$ Can we emulate this somehow?

Problem: BP breaks down for multiple hidden layers – vanishing gradients due to saturation of sigmoids (actually maybe not so true . . .).

Recent research [Hinton et al., 2006] uses unsupervised pre-training by Restricted Boltzmann Machines to get into vicinity of a good minimum.

Deep Autoencoder Example: Face Compression



Deep Autoencoder Example: Text Categorization

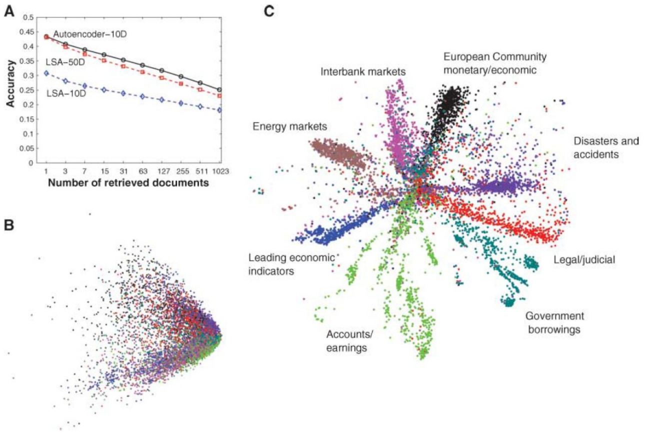


Fig. 4. (**A**) The fraction of retrieved documents in the same class as the query when a query document from the test set is used to retrieve other test set documents, averaged over all 402,207 possible queries. (**B**) The codes produced by two-dimensional LSA. (**C**) The codes produced by a 2000-500-250-125-2 autoencoder.

Excursion: Unsupervised Learning

Goal: Find a P-dim representation (=feature vector) that describes the N-dim input vector $\mathbf{x}$, where usually $N > P$ or $N \gg P$.

Advantages:

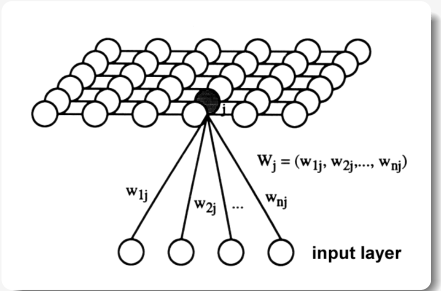
- reduced redundancy may(!) lead to better classification and generalization in subsequent processing
- maximum information transmission through compact representation
- decorrelated features lead to numerical speed-ups (diagonal matrices!)
- prediction of non-trivial properties of input patterns

Problem: Nobody to tell us where in the input data the information is found.

Self-Organizing Maps (SOMs)

An SOM consist of an input layer of arbitrary dimension, and a hidden layer that is usually 1d, 2d, or 3d.

The hidden layer imposes a fixed proximity relationship between the neurons.



Idea: Each neuron to represent a certain typical input vector.
$\Longrightarrow$ Can define similarity of those input vectors by looking at the proximity map of the hidden neurons!

Training SOMs

1. Fetch next input pattern $\mathbf{x} \in \mathbb{R}^n$.
2. Compare input vector against weights $\mathbf{W}_j = (w_{1j}, \dots, w_{nj}$ of all hidden neurons j using any metric, to find the best match ("winner") m. Typical metrics are:

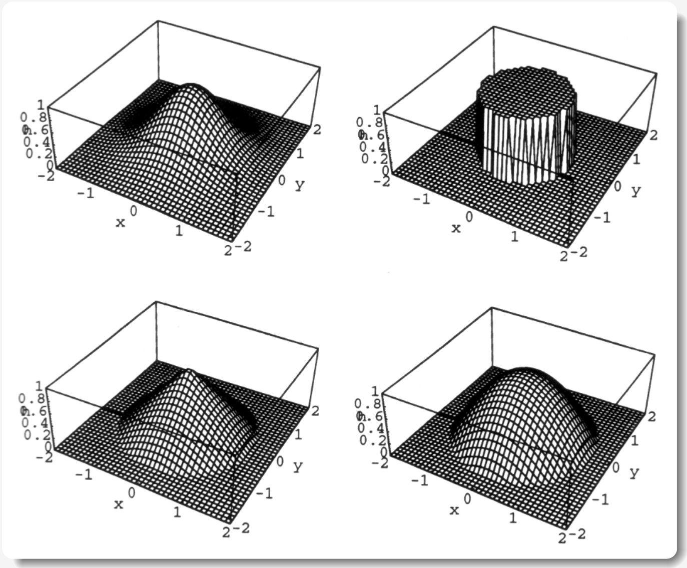
$$m = \text{argmin}_j \|\mathbf{x} - \mathbf{W}_j\| \qquad \text{or} \qquad m = \text{argmax}_j (\mathbf{x} \cdot \mathbf{W}_j)$$

3. Adjust weights according to

$$\mathbf{W}_j^{(\tau+1)} = \mathbf{W}_j^{(\tau)} + \eta^{(\tau)} K_{mj}^{(\tau)} (\mathbf{x}^{(\tau)} - \mathbf{W}_j^{(\tau)})$$

4. Decrease learn rate η and/or width of kernel function K_{mj} with timestep τ to force convergence.
5. If not converged, go to 1.

Typical SOM kernel functions


1
0.8
0.6
0.4
0.2
0
-2
-1
0
x
1
2
-2
-1
0
y
1
2